INTERDISZIPLINÄRES ZENTRUM
FÜR WISSENSCHAFTLICHES RECHNEN

SONDERFORSCHUNGSBEREICH 359
REAKTIVE STRÖMUNGEN, DIFFUSION UND TRANSPORT

FEMLISP - a tool box for solving partial differential equations with finite elements and multigrid

Nicolas Neuß

Preprint 2003 - 11 (SFB 359)

August 2003

UNIVERSITÄT HEIDELBERG

Ruprecht-Karls-Universität Heidelberg

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen

Sonderforschungsbereich 359
Reaktive Strömungen, Diffusion und Transport

FEMLISP - a tool box for solving partial differential equations with finite elements and multigrid

Nicolas Neuß

Preprint 2003 - 11 (SFB 359) August 2003

Diese Arbeit wurde von der Deutschen Forschungsgemeinschaft gefördert

eingereicht bei: *Computing and Visualization in Science*

Ruprecht-Karls-Universität Heidelberg

Interdisziplinäres Zentrum für
Wissenschaftliches Rechnen

Sonderforschungsbereich 359
Reaktive Strömungen, Diffusion und Transport

Im Neuenheimer Feld 294
D-69120 Heidelberg

Telefon:	+49 (6221) 54-8977
Telefax:	+49 (6221) 54-8652
e-mail	sfb359@iwr.uni-heidelberg.de
URL	http://www.sfb359.uni-heidelberg.de/

FEMLISP— a tool box for solving partial differential equations with finite elements and multigrid

Nicolas Neuß

July 23, 2003

Abstract

In this article, we present the tool box FEMLISP for solving partial differential equations. FEMLISP uses the finite element method on unstructured meshes of arbitrary space dimensions and uses several types of multigrid algorithms for solving the arising linear systems. It is written in Common Lisp which has several important benefits. First, FEMLISP is interactive without the need of an additional scripting language. Second, the high expressiveness of Lisp leads to a very compact repesentation of the underlying mathematical ideas.

1 Introduction

This article is concerned with software for solving partial differential equations (PDEs). The theoretical and practical treatment of partial differential equations is a highly complex domain of mathematics with a multitude of possible phenomena. Consequently, the design of optimal numerical methods is also difficult, and the efficient implementation of such methods is an even more challenging problem. Therefore, originally, software in this domain could handle only very special problems well and was not applicable to other situations. But this changed in the last decade, when more and more software packages appeared, claiming to be multi-purpose tools for solving partial differential equations.

This paper presents such a multi-purpose tool. But in contrast to other approaches, which are mostly written using machine-oriented languages like Fortran, C, or C++, our application is written in Lisp, which is a language long known from research in artificial intelligence.

Lisp is the second-oldest high-level computer language after Fortran. It is a very flexible and powerful language, and a lot of problems arising in computational science were first solved using Lisp as a vehicle. Besides being well-suited for solving problems in artificial intelligence (AI), the world's first computer algebra system, *Macsyma*, was implemented in Lisp, and, more recently, Lisp was used in the package *Kenzo* to solve difficult problems in algebraic topology. Lisp is also the language of *Autocad* and the editor *Emacs*. Astonishingly, up to now, Lisp has not been a mainstream language. The reasons for this are mostly historical: in the first decades of computer history, resources were scarce and a language for which the first implementations were interpreters, which has automatic memory management as an essential component, and which is used best inside large development environments could not compete with lightweight languages for the computationally easy problems which were tackled at that time.

Nowadays, the situation is drastically different. Computing power has increased tremendously, thus making it possible to have powerful Lisp environments on personal computers while still using only a fraction of the available memory. Due to Java, automatic memory management has become a mainstream feature. In the meantime, Lisp itself has grown up into *Common Lisp*, which is a powerful object-oriented language for handling real-world applications, and for which many implementations support native code compilation.

Consequently, using Lisp for problems outside its original realm of artificial intelligence is a very natural choice today, see [7], [6], [12]. Comparing FEMLISP [8] with other PDE toolboxes, we observe the following advantages which cannot be easily obtained using conventional languages like Fortran, C, or C++:

1. FEMLISP is interactive. Toolboxes written in conventional languages have to implement this feature themselves or to be linked with some scripting language like Perl or Python. In any case, an artificial interface between scripting level and application language appears, which is not easy to maintain.

2. The editor is integrated in a way that is very difficult to achieve for conventional languages. For example, argument lists of functions appear when typing, and the documentation of functions and variables is available at a keystroke. Also, the debug cycle is as short as can possibly be: changing a function, recompiling it separately, and testing it in the running environment is usually a matter of only a few keystrokes and fractions of a second runtime.

3. Lisp is very expressive, and programming techniques can be used which would introduce a large overhead in conventional languages (some examples are given in Section 6). This leads to a concise representation of the underlying mathematical ideas and therefore to a very short and maintainable source code. To achieve maximum benefit, the source code is available under a liberal open source license.

The structure of this article is as follows: in Section 2, we list features which a multi-purpose tool for solving PDEs should have. Next, Section 3 gives an overview of FEMLISP's features and their implementation, focusing on certain points where FEMLISP deviates from other approaches. Section 6 shows an application of FEMLISP to a certain problem. Finally, in Section 7, we discuss how well FEMLISP meets the requirements posed at the beginning and what remains to be done.

2 PDE software requirements

The following is a list of properties which we think important for a multi-purpose tool box for solving partial differential equations. All of them are fundamental and non-trivial to implement.

1. Arbitrary domains $\Omega \subset \mathbb{R}^n$ can be handled, with n being at least 1,2, and 3. In general, for $n > 1$, a good approximation of arbitrary domains Ω will require either the use of unstructured grids or the use of similarly complex constructions like locally smooth patches of structured grids. Note that other values of n are also interesting. The case $n = 0$ represents ordinary differential equations, and the case $n = 4$ occurs, for example, when solving problems in space-time in general relativity.

2. On subregions where the solution is smooth, approximations of higher order are necessary for an optimal approximation. Additionally, the boundary of the domain should also be approximated with the same order of accuracy. For smooth problems and modest-to-high accuracy requirements, this feature can be considered a necessity.

3. If the solution has distinct local features, grid adaptivity (h-adaptivity) is necessary. In analogy to the previous point, there are problems in practice where this is the most important quality a PDE solver should have. An important ingredient here is good error estimators.

4. Usually, the solution of linear system appears at some place when solving problems in numerical analysis. In the case of finite element methods, the linear systems arising are very large and additionally quite sparse, because the support of basis functions typically intersects only a few elements. For such systems, the application of direct solvers is suboptimal or even impossible on current hardware. Thus, it is important for fast solvers to be available, especially iterative methods with hierarchical preconditioners, e.g. multigrid.

5. Unstructured grids, local adaptivity and multigrid are rather complex from a software engineering viewpoint and introduce a large administrative overhead in the implementation. It is possible to eliminate this overhead in most situations; yet, up to now only very special cases have been optimized in this way.

6. Parallelization is essential for handling real-world applications involving huge amounts of data.

7. Interactivity is a necessary requirement for user-friendly software in almost any field of applications, and it is also a very convenient feature for the developer. Unfortunately, most academic software for solving PDEs is not interactive. The rest usually achieves interactivity by linking to some scripting language, e.g. Perl, Python, or TCL. Unfortunately, this approach is suboptimal. First, programmer and/or user have to learn an additional language. Second, this approach introduces an artificial interface between scripting and application language which is not easy to maintain. [1]

8. An important point which is often neglected is the quality of the software from a software engineering viewpoint. Especially, the source code should be compact and easy to understand (thus allowing for fast modifications) while still being general and extensible. At first glance, these may appear to be contradictory goals, but experience shows that reasonable compromises can usually be found, provided that the right formulation is chosen. In any case, they depend heavily on the expressivity of the computer language in use. Note also that this quality can only be determined if the source code is available; this rules out some commercial software.

We think that it is very reasonable to judge PDE software by how well it meets the requirements above *in an integrated way.* It is not sufficient if only one or two of these properties are available, or if certain properties are implemented in an incompatible way (i.e. there are separate versions of the same program satisfying different properties of the above list).

[1]Commercial packages usually offer interactivity by means of a graphical user interface (GUI), which can be regarded as a very simple scripting language. Nevertheless, it is a different situation, because many users require exactly such a GUI and do not want to be bothered with either application or scripting language. The main problem here is that the GUI is usually too limited for sophisticated users, so that an additional interactive scripting language level becomes necessary, see e.g. [5].

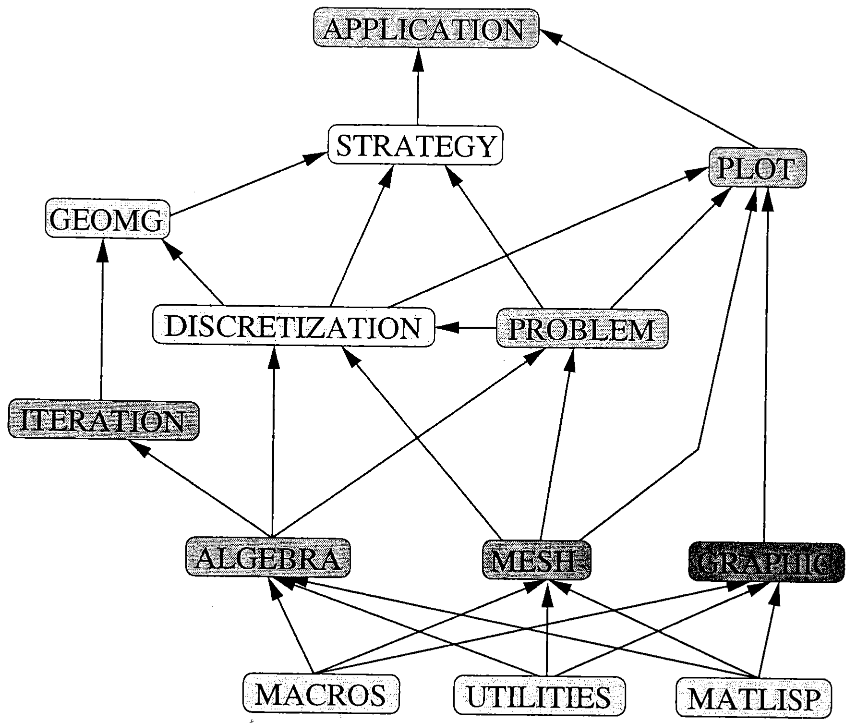

Figure 1: Important FEMLISP packages and their dependency.

3 Femlisp overview

In brief, FEMLISP [8] has the following features:

- Interactive environment
- Unstructured grids in arbitrary space dimensions $n \geq 0$. Elements can be arbitrary products of simplices.
- Isoparametric and non-parametric element mappings
- Local grid refinement
- Lagrange finite elements of arbitrary order
- Geometric and algebraic multigrid methods
- Graphics (using *Gnuplot*, *Data Explorer*)

Fig. 1 shows the internal structure of Femlisp. It is grouped in modules with module dependency indicated by arrows. The modules appear in this form in the project file `start.lisp`. They usually correspond to a Common Lisp package, but may sometimes comprise several packages, where a package defines a namespace in Common Lisp.

The modules shown in this figure can be roughly ordered in levels. The lowest level consists of the three modules `MACROS`, `UTILITIES`, and `MATLISP`. `MACROS` and `UTILITIES` extend CL with some useful macro and function definitions, while `MATLISP` contains a CL interface [11] to the Fortran `BLAS` and `LINPACK` routines [4].

The second level consists of the modules `ALGEBRA`, `MESH` and `GRAPHIC`. `GRAPHIC` provides a low-level interface to external graphic software; at the moment both IBM's *OpenDX* and *Gnuplot* are supported. `ALGEBRA` contains the data structures and operations used for linear algebra and `MESH` contains mesh management including domain definitions. Both the `MESH` and `ALGEBRA` module will be discussed in more detail below.

The third level consists of the following modules:

1. The `ITERATION` module which includes the definition for the abstract classes `<solver>`, `iteration`, as well as the generic function `solve` which constitute the interface for linear and non-linear solving. Several instances of these classes are implemented, including the conjugate gradient iteration and algebraic multigrid (AMG). The module also contains the `GEOMG` package, which handles iterations that depend on geometric information, e.g. from the discretization. At the moment, these are the geometric multigrid iteration, an AMG-like scheme for preconditioning high-order discretizations with low-order ones, and some smoothers of Vanka type.

2. The `DISCRETIZATION` module defines `<discretization>` as an abstract class and `<fe-discretization>` as a concrete derived class. A generic function `get-fe` is used for associating a cell with a *finite element* `<fe>`, which is a data structure containing information about base functions and dual functionals on the corresponding cell. Lagrange finite elements of arbitrary order are implemented as a special instance of `<fe-discretization>`. Note that other discretizations as finite differences or finite volumes could easily be incorporated as well.

3. The `PROBLEM` module introduces the `<problem>` class and methods. Several derived problems are defined, e.g. `<cdr-problem>` for convection-diffusion-reaction problems, `<elasticity>` for elasticity problems, and `<navier-stokes>` for Navier-Stokes problems.

The third level provides another level of abstraction. It consists of the modules `STRATEGY` and `PLOT`. `STRATEGY` provides methods for solving problems by adaptive FEM, and PLOT defines generic functions and methods for post-processing (plotting of coefficients, meshes, and functions).

The fourth level APPLICATION has access to a lot of basic modules, especially STRATEGY, DISCRETIZATION, and PLOT. There are several separate directories and files containing applications of FEMLISP to special problems.

In Section 5, some of these modules will be discussed in more detail.

4 Advanced programming techniques

In FEMLISP, some advanced programming techniques are used which may seem unusual to programmers used to other languages. We briefly explain some of them. For more information, see [14].

4.1 Object-oriented programming

FEMLISP is implemented largely in an object-oriented manner using CLOS (*Common Lisp Object System*). CLOS is very powerful, featuring, for example, multiple inheritance, multi-argument dispatch, and class-redefinition at runtime. We do not want to go into detail here, but refer to the books [9] and [10]. Nevertheless, we want to discuss briefly some specialities of object-oriented programming in Common Lisp which are not common in other programming languages.

Dispatch of a generic function on more than one argument is often useful. One example is the following code for a matrix-vector multiplication taken from `sparse.lisp`, where the method cannot be assigned clearly to either the class `<matrix>` or the class `<vector>`.

```
(defmethod m* ((A <matrix>) (y <vector>))
  (let ((x (make-row-vector-for A)))
    (x+=Ay x A y)
    x))
```

Method modification opens up a nice way for enhanced code re-use by allowing a method for a derived class to modify methods for superclasses by defining `:before`, `:after`, and `:around` methods. For example, the following modifying method adds a compatibility check to the above matrix-vector multiplication.

```
(defmethod m* :before (A y)
  (assert (= (ncols A) (length y))))
```

Multiple inheritance is usually not used as much as single inheritance, so that some object-oriented programming languages, for example Java, do not provide it at all. Nevertheless, it can be quite useful. In FEMLISP, it is used, for example, to define so-called *mixin classes* which are used to dispatch the behaviour of a multigrid scheme between the standard *correction scheme* (CS) and Brandt's *full approximation scheme* (FAS), see the code at the end of Subsection 5.5.

Class redefinition at runtime is very useful, especially in the development phase of a program, since the same Lisp session is often used during several days or even weeks.

4.2 Memoization

Quite often it happens that some time-intensive function is called very frequently on only few values. Easy examples of this are simple recursive processes as the computation of binomial or Fibonacci numbers. In FEMLISP, this occurs at several places, e.g. when computing n-dimensional reference elements and corresponding refinement rules, when computing finite element data for those reference elements, and so on. A useful technique for accelerating programs in this case is so-called "memoization", which means storing already computed values in a table and retrieving them from there if possible.

Now, in Common Lisp, it is easy to construct a language extension which turns an existing function into a memoized one, see e.g. [14]. The code is simply the following:

```
(defun memoize-symbol (funsym)
  "Memoizes the function named by the given symbol."
  (let ((unmemoized (symbol-function funsym))
        (table (make-hash-table :test #'equalp)))
    (setf (symbol-function funsym)
          #'(lambda (&rest args)
              (multiple-value-bind (value found)
                  (gethash args table)
                (if found
                    value
                    (setf (gethash args table)
                          (apply unmemoized args))))))))
```

An application of this technique to the recursive computation of Fibonacci numbers then yields something like the following:

```
* (defun fib (n)
  (case n
    (0 0) (1 1)
    (t (+ (fib (- n 1))
      (fib (- n 2))))))
FIB
* (time (fib 35))
; Evaluation took:
;   1.31 seconds of real time
;   1.3 seconds of user run time
;   0.0 seconds of system run time
;   0 page faults and
;   0 bytes consed.
;
9227465
```

```
* (memoize-symbol 'fib)

* (time (fib 35))
; Evaluation took:
;   0.0 seconds of real time
;   0.0 seconds of user run time
;   0.0 seconds of system run time
;   0 page faults and
;   552 bytes consed.
;
9227465
```

4.3 Flexible argument lists and assembly lines

In machine-oriented languages, it is usually required that both the number of function arguments and their types are known and, as soon as functions need a larger number of parameters, these are collected in some structure or class. Because this is quite inflexible, functions in Common Lisp are allowed to have a variable number of parameters of arbitrary type. Additionally, the arguments can be named with certain keywords. This means that the keyword part of the argument list has an even number of elements which appear in pairs of the form *symbol – value* (such a list is called a *property list*). This allows for a very flexible parametrization of a lot of operations.

In FEMLISP, a variant of this technique has proven useful in several places. Here, instead of a standard result, the function adds its output to a property list handed to the function as a parameter. We call such a property list an "assembly line". This has several benefits:

1. The function can grab arbitrary parameters from the assembly line.
2. A function can put (named) values on the assembly line.
3. The assembly line can then be passed to other functions.

Such assembly lines are used, for example, in the implementation of the Stüben AMG (algebraic multigrid) algorithm in `stueben.lisp`, and in the implementation of the problem-solving strategy in `strategy.lisp`. For example, the default strategy for solving problems reads as follows:

```
(defmethod solve-with ((strategy <strategy>) (problem <problem>)
                       assembly-line)
  (loop (sufficient-p strategy assembly-line)
        (when (getf assembly-line :end-p) (return assembly-line))
        (improve-guess strategy assembly-line)))
```

The methods `initial-guess` and `improve-guess` specialized for the class `<fe-strategy>` are then in turn composed from assembly-line operations performing discretization, solution of the linear system, error estimation, and mesh adaption.

4.4 Integrated documentation, testing, and demos

Common Lisp offers several interesting possibilities for integrating the documentation and testing phase with programming in a way which cannot easily be done in languages that are not interactive or do not have sufficient introspection features.

First, function and variable declarations may contain docstrings documenting their use. These strings can be asked for in the interactive environment. There are also tools which extract those docstrings for composing a reference manual. This is no substitute for a good user manual, of course, but an important supplement.

Second, regression test suites can be constructed easily. In FEMLISP, this is done by putting a test function at the end of most files. This function then checks several critical features which the file or module provides. It is added to the test suite by the command `adjoin-femlisp-test`. Then, after loading FEMLISP, all these functions can be executed with the command `run-femlisp-tests`. Errors and exceptions are registered and reported later on.

Third, a demo suite is also built into FEMLISP in a similarly distributed manner. Wherever something interesting could be demonstrated, a small demo node is generated with `make-demo` and added to the tree of all demos with `adjoin-demo`. After loading FEMLISP, the whole demo suite is available and can be run with the command `demo`.

5 Details of some Femlisp modules

In the following subsections, we discuss some of the modules in more detail. A note on the syntax: we have mostly used class names of the form `<...>`. This makes the programs more readable, even if it differs from the notation used for built-in classes.

5.1 The `ALGEBRA` module

This module contains definitions for doing linear algebra in FEMLISP and consists of several files. Besides others, these are `vector.lisp` and `matrix.lisp`, where an abstract interface for linear algebra on vectors and matrices is defined. In `matlisp.lisp`, this interface is realised for Matlisp matrices, and in addition the matlisp is partially extended to arrays. In `crs.lisp`, the well-known compact row-ordered storage is defined for storing sparse matrices, `tensor.lisp` contains class and method definitions for full tensors of arbitrary rank.

The file `sparse.lisp` then introduces a sparse storage scheme for block vectors and block matrices over arbitrary index sets. This is very convenient for functions and linear operators defined on unstructured grids because the geometric grid objects themselves can index their degrees of freedom. It also allows for local updates when advanced adaptive schemes like the one proposed in [15] are used.

The definition of the classes `<sparse-vector>` and `<sparse-matrix>` is as follows

```
(defclass <sparse-vector> ()
  ((blocks :initform (make-hash-table) :type hash-table)
   (key->size :reader key->size :initarg :key->size
              :type (function (t) positive-fixnum))
   (multiplicity :reader multiplicity :initform 1
                 :initarg :multiplicity :type fixnum)
   (print-key :reader print-key :initarg :print-key
              :initform #'princ :type (function (t))))
  (:documentation "The slot blocks contains a hash-table of vector blocks
indexed by keys.  The function key->size determines the block size, the
function print-key determines how each key is printed.  Multiplicity is
used for handling multiple right-hand sides/solutions simultaneously."))

(defclass <sparse-matrix> ()
  ((row-table :accessor row-table :initarg :row-table
              :initform (make-hash-table) :type hash-table)
   (column-table :accessor column-table :initarg :column-table
                 :initform (make-hash-table) :type hash-table)
   (print-row-key :reader print-row-key :initarg :print-row-key
                  :initform #'princ :type function)
   (print-col-key :reader print-col-key :initarg :print-col-key
                  :initform #'princ :type function)
   (row-key->size :reader row-key->size :initarg :row-key->size
                  :type (function (t) positive-fixnum))
   (col-key->size :reader col-key->size :initarg :col-key->size
                  :type (function (t) positive-fixnum))
   (keys->pattern :reader keys->pattern :initarg :keys->pattern
                  :type function))
  (:documentation "The <sparse-matrix> represents an unordered matrix
graph."))
```

Basic methods for those classes are also defined in `sparse.lisp`. An LU decomposition for `<sparse-matrix>` is implemented in `sparselu.lisp`.

Unfortunately, the use of hash-tables instead of arrays is much slower than working with, for example, compact row-ordered storage. Thus, good performance can only be expected if the inner blocks are relatively large. This is the case for systems of equations and/or approximations of higher order. Future versions of FEMLISP will probably improve on that by using an array-based scheme similar to the one in `sparse-tensor.lisp`. However, to obtain a speed comparable to statically typed languages as C++, it will also be necessary to avoid the type dispatch for operations on the blocks (which can in principle be vectors/matrices of any kind). This can be done by compiling type-adapted code for the cases encountered frequently.

5.2 The MESH module

This module contains the definitions of meshes and routines for mesh management. The meshes allowed in FEMLISP are more general than those of most other software for solving PDEs. In FEMLISP, both mesh, domain and problem definitions are defined over an underlying abstraction, the so-called `<skeleton>`. A `<skeleton>` captures the mathematical idea of a *cell complex* which builds a topological space by mapping from standard domains in $\mathbb{R}^n$. Now, a `<skeleton>` can be seen as mapping the cells of such a cell complex to arbitrary values. Then, a `<domain>` is a `<skeleton>` where each cell (which we call *patch* in this case) is mapped to geometric properties, and a `<mesh>` is a `<skeleton>` where each cell is mapped to the domain patch to which it belongs.

The basic definitions of this module are

```
(defclass <cell> ()
  ((cell-class :reader cell-class :initarg :cell-class
               :type <cell-class>)
   (boundary :reader boundary :initarg :boundary :initform #()
               :type cell-vec)
   (mapping :reader mapping :initarg :mapping nil))
  ;;
  (:documentation "The basic cell class.  Every cell consists of its
class collecting all class information, an array of boundary cells and a
mapping slot.  That slot contains the position for vertices and a possibly
nonlinear mapping for other cells.  A value of nil means that multilinear
interpolation between the corners is used for constructing the mapping."))
```

```
(defclass <skeleton> ()
  ((dim :accessor dimension :initarg :dimension :type (integer -1))
   (etables :accessor etables :type (array t (*))))
  ;;
  (:documentation "A skeleton is a vector of hash-tables containing the
cells of a certain dimension as keys.  The information stored in the
values is different depending on the subclass derived from skeleton."))

(defclass <domain> (<skeleton>)
  ((boundary :accessor domain-boundary))
  (:documentation "A <domain> is a special <skeleton>.  Its cells are
called patches, and the values are property lists carrying geometric
information, e.g. metric, volume-form, embedding or identification."))

(defclass <mesh> (<skeleton>)
  ((domain :accessor domain :initarg :domain :type <domain>)
   (parametric :accessor parametric :initform nil :initarg :parametric))
  ;;
  (:documentation "A <mesh> is a special <skeleton> mapping cells to
property lists with properties of the cell.  The most important property of
a cell is its patch in the domain.  Another one could be a list of possibly
identified cells.  The slot parametric determines which kind of cell
mappings are used for approximating the domain.  These can be the nonlinear
mappings used in the domain definition, but also arbitrary approximations,
to those mappings, e.g. isoparametric mappings.  The special value NIL
means that multilinear mappings are used for all cells outside the
boundaries."))

(defclass <hierarchical-mesh> (<mesh>)
  ((levels :accessor levels :initarg :levels
           :type (array <skeleton> (*))))
  (:documentation "Hierarchical-meshes are those meshes which will be used
most often, because they remember the refinement history and therefore
allow for refinement and coarsening.  The slot levels is an array of
skeletons containing the cells for different levels."))
```

Meshes can be refined either uniformly or locally using the Freudenthal algorithm as presented in [2] and generalized to product elements. When local refinement is used, hanging nodes may occur. In contrast to most other finite element software, in FEMLISP the difference of refinement levels of adjacent cells may be arbitrarily large. Up to now, anisotropic refinement of tensorial cells has not yet been implemented.

5.3 The PROBLEM module

The problem module consists of the package PROBLEM together with packages for special problems like convection-diffusion-reaction equations, elasticity or Navier-Stokes.

The PROBLEM package defines especially the following basic interface:

```
(defclass <problem> ()
  ((domain :accessor domain :initform (ext:required-argument)
           :initarg :domain :type <domain>)
   (p->c :accessor patch->coefficients :initform (ext:required-argument)
         :initarg :patch->coefficients)
   (memoize :initform t :initarg :memoize)
   (multiplicity :reader multiplicity :initform 1 :initarg :multiplicity))
  ;;
  (:documentation "Base-class for a pde-problem.  The domain slot contains
the domain on which the problem lives.  The p->c slot contains a map from
the domain patches to problem coefficients.  Those are property lists of
the form (:NAME1 coefficient1 :NAME2 coefficient2 ...).  The multiplicity
slot can be chosen as n>1 if the problem is posed with n different right
hand sides simultaneously."))

(defgeneric coefficients (problem)
  (:documentation "Yields a list of possible coefficients for problem."))

(defclass <coefficient-input> ()
  ((local :reader ci-local :initarg :local :initform nil :type t)
   (global :reader ci-global :initarg :global :initform nil :type t)
   (solution :reader ci-solution :initarg :solution
             :initform nil :type t))
  (:documentation "The <coefficient-input>-class represents the interface
between discretization and problem.  It may be extended as needed, e.g. to
allow for coefficients depending on the solution gradient.  This class is
also used to construct a sample input for a <coefficient> by giving the
needed entries the value t or nil."))

(defclass <coefficient> ()
  ((input :accessor sample-input :initarg :input
          :type <coefficient-input>)
   (eval :accessor coeff-eval :initarg :eval
         :type function))
  (:documentation "A class for coefficient-functions.  input is a sample
input indicating the needed/non-needed fields with a true resp. false
value.  eval is the evaluating function."))

(defmethod evaluate ((coeff <coefficient>) (ci <coefficient-input>))
  "The pairing between coefficient function and input."
  (funcall (coeff-eval coeff) ci))
```

Then, for example, the Navier-Stokes problem in `navier-stokes.lisp` is derived as follows:

```
(defclass <navier-stokes-problem> (<problem>)
  ()
  (:documentation "Navier-Stokes problem."))

(defmethod coefficients ((problem <navier-stokes-problem>))
  "Coefficients for the Navier-Stokes problem."
  '(VISCOSITY REYNOLDS FORCE CONSTRAINT))
```

Several sample problems, for example the well-known *driven cavity*, are defined in this file as well.

5.4 The `DISCRETIZATION` module

This module contains the following basic definitions for finite element discretizations in `fe.lisp`:

```
(defclass <discretization> ()
  ()
  (:documentation "Discretization base class."))

(defclass <fe-discretization> (<discretization>)
  ((nr-comps :reader nr-of-components :initform 1 :initarg :nr-comps))
  (:documentation "The base class for fe discretizations."))

(defgeneric get-fe (fe-disc cell)
  (:documentation "Returns the finite element for the given discretization
and reference cell."))

(defclass <standard-fe-discretization> (<fe-discretization>)
  ((order :reader discretization-order :initarg :order)
   (cell->fe :initarg :cell->fe))
  (:documentation "For this class the finite elements are obtained from a
cell->fe mapping given as a class slot.  Also the order is predetermined,
thus excluding hp-methods."))
```

Obviously, there are possibilities for incorporating non-standard finite element discretizations like hp-methods as well, or even very different discretizations like finite-volume or finite-difference schemes into this interface. The key for local assembly is given by the generic function `get-fe`, which yields a suitable finite element for a given cell. The value of `get-fe` is a class `<fe>` for scalar problems or `<vector-fe>` for vector-valued problems which contains information on base functions and node functionals. Another generic function `quadrature-rule` computes memoized quadrature rules for those finite elements.

In the file `fedisc.lisp`, the function `fe-discretize` is defined. This function performs the standard steps for finite element discretization: interior assembly, boundary-constraint assembly, hanging- node-constraint assembly, and the ensuing constraint elimination. It works on an *assembly line* as explained in Section 4.3.

5.5 The ITERATION module

This module consists of several packages. Basic concepts of iterations and solvers are defined in the package `ITERATION`. An extract of the interface looks as follows:

```
(defclass <iteration> ()
  ((damp :reader damping-factor :initform 1.0 :initarg :damp)
   (output :reader output :initform nil :initarg :output))
  (:documentation "The <iteration> base class."))

(defclass <linear-iteration> (<iteration>)
  ()
  (:documentation "The <linear-iteration> class.  Linear iterations are
e.g. <gauss-seidel>, <cg>, or <multigrid>."))

(defclass <iterator> ()
  ((matrix :reader matrix :initarg :matrix)
   (initialize :reader initialize :initarg :initialize :initform nil)
   (iterate :reader iterate :initarg :iterate :type function)
   (residual-before :reader residual-before :initarg :residual-before)
   (residual-after :reader residual-after :initarg :residual-after))
  ;;
  (:documentation "An <iterator> object contains functions doing iteration
work or flags indicating which work has or has not to be done for calling
that iterator.  It is generated by the generic function make-iterator."))

(defgeneric make-iterator (linit mat)
  (:documentation "Constructs an iterator object given a linear iteration
and a matrix."))

(defclass <solver> ()
  ((maxsteps :reader maxsteps :initform nil :initarg :maxsteps)
   (threshold :reader threshold :initform nil :initarg :threshold)
   (reduction :reader reduction :initform nil :initarg :reduction)
   (residual-norm :reader residual-norm :initform #'norm
                  :initarg :residual-norm)
   (output :reader output :initform nil :initarg :output))
  (:documentation "The base class of linear, nonlinear and whatever
iterative solvers."))

(defgeneric solve (solver &rest parameters)
  (:documentation "Solve a problem specified through the parameter list."))
```

Several standard iterations are available, e.g. Gauss-Seidel, SOR, ILU (in `linit.lisp`) and CG (in `krylow.lisp`). A larger block of code is contained in a separate package `MULTIGRID` and contains the multigrid iteration. The basic definition of the multigrid iterations is the following (see the file `multigrid.lisp`):

```
(defclass <mg-iteration> (<linear-iteration>)
  ((pre-smooth :reader pre-smooth :initform *default-smoother*
               :initarg :pre-smooth)
   (pre-steps :reader pre-steps :initform 1 :initarg :pre-steps)
   (post-smooth :reader post-smooth :initform *default-smoother*
                :initarg :post-smooth)
   (post-steps :reader post-steps :initform 1 :initarg :post-steps)
   (gamma :reader gamma :initform 1 :initarg :gamma)
   (base-level :reader base-level :initform 0 :initarg :base-level)
   (coarse-grid-iteration :reader coarse-grid-iteration
                          :initform *default-coarse-grid-iteration*
                          :initarg :coarse-grid-iteration)
   (fmg :reader fmg :initform nil :initarg :fmg))
  (:documentation "The multigrid iteration is a linear iteration specially
suited for the solution of systems of equations incorporating elliptic
terms.  In ideal situations, it solves such systems with optimal
complexity.  It is a complicated linear iteration, which consists of
applying simple linear iterators as smoothers on a hierarchy of grids.
This grid hierarchy is obtained either by discretizing on successively
refined meshes (geometric multigrid) or it is constructed from matrix
information alone (algebraic multigrid).

The <mg-iteration> is not intended to be used directly.  Incorporating
mixins like <correction-scheme> or <fas> results in concrete classes like
<algebraic-mg>."))
```

```
(defclass <correction-scheme> ()
  ()
  (:documentation "This is a mixin-class which yields the correction scheme
variant of multigrid."))

(defclass <fas> ()
  ()
  (:documentation "This is a mixin-class for <mg-iteration> which yields
the behaviour of Brandt's FAS scheme."))
```

From this class, an algebraic multigrid iteration is derived in `amg.lisp` and a geometric multigrid iteration in `geomg.lisp`. For example, the definition for algebraic multigrid is the following:

```
(defclass <algebraic-mg> (<correction-scheme> <mg-iteration>)
  ((max-depth :reader max-depth :initform most-positive-fixnum
              :initarg :max-depth)
   (cg-max-size :reader cg-max-size :initform 1
                :initarg :cg-max-size))
  (:documentation "The algebraic multigrid iteration is a multigrid
iteration where the hierarchy of problems is derived from the fine-grid
matrix.  Usually, an algebraic multigrid will use the same iterator as its
geometric counterpart."))
```

This definition is then refined further towards algebraic multigrid methods of selection and aggregation type. Those are in turn refined towards variants, for example the Ruge-Stüben method described in [16], [17].

6 A sample application

FEMLISP was used for computing effective coefficients for heterogeneous media in several situations with periodically arranged heterogeneities. These calculations include effective diffusion coefficients, effective elasticity tensors, and effective boundary layer constants for oscillating boundaries, see [13]. Here, we include results for the computation of an effective elasticity tensor for a porous structure.

The geometry of the representative cell Y is obtained as

$$Y = (0,1)^d \setminus B\,, \text{ where} B := \{y \mid (\sum_{i=1}^{d} |y_i - \frac{1}{2}|^2)^{1/2}\}\,. \tag{1}$$

Let now $A_{ij}^{kl} : Y \to \mathbb{R},\ i,j,k,l \in \{1,\ldots,n\}$ be the coefficients of a tensor of rank 4 satisfying the symmetry conditions

$$A_{ij}^{kl}(y) = A_{ji}^{lk}(y) = A_{kj}^{il}(y) \tag{2}$$

for all $i,j,k,l \in \{1,\ldots,n\}$ and all $y \in Y$. Further, we require ellipticity and continuity in the sense that

$$\eta_{ik}\eta_{ik} \lesssim A_{ij}^{kl}(y)\eta_{ik}\eta_{jl} \lesssim \eta_{ik}\eta_{ik} \tag{3}$$

for all $y \in Y$ and all symmetric matrices $\eta = (\eta_{ij})_{i,j=1,\ldots,n}$ where $a \lesssim b$ means $a \le Cb$ with some moderate constant C.

Following [1], the effective elasticity tensor is then computed by

$$\hat{A}_{iq}^{kr} = \int_Y A_{ij}^{kl}(y)\big(\delta_{jq}\delta_{lr} + \frac{\partial N_q^{lr}}{\partial y_j}(y)\big)\,dy \tag{4}$$

where N is a tensor of rank 3 satisfying the cell problem

$$-\frac{\partial}{\partial y_i}\big(A_{ij}^{kl}(y)(\delta_{jq}\delta_{lr} + \frac{\partial N_q^{lr}}{\partial y_j}(y))\big) = 0\,, \qquad y \in Y\,. \tag{5}$$

For our numerical test, we look at the special case of a constant isotropic elasticity tensor

$$A_{ij}^{kl}(y) = A_{ij}^{kl} = \lambda\delta_{ik}\delta_{jl} + \mu(\delta_{ij}\delta_{kl} + \delta_{kj}\delta_{il}) \tag{6}$$

with the Lamé constants $\lambda = \mu = 1$. First, we consider the two-dimensional case. Since the solution is smooth in Y, it makes sense to use high-order finite element approximations, provided that we approximate the domain

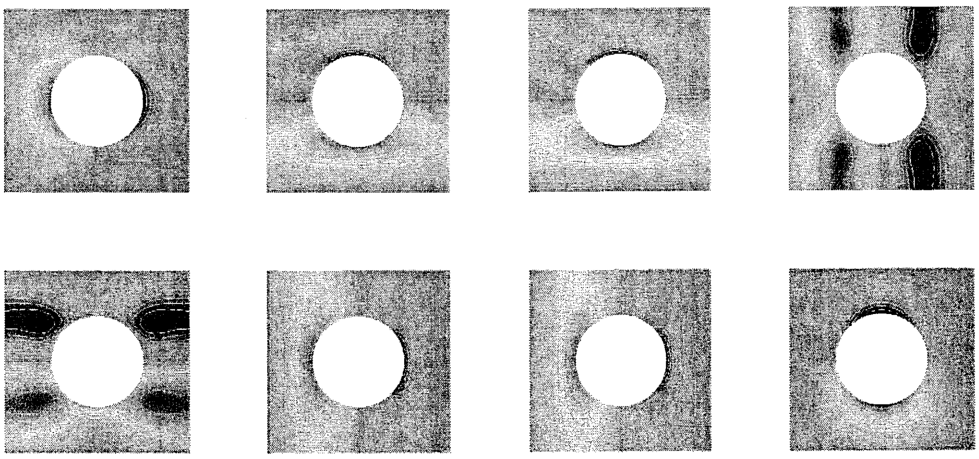

Figure 2: The eight components of the tensor N_q^{lr}.

sufficiently well. For the simple geometry considered here, we have chosen the (nonlinear) cell mappings in such a way that they cover the domain Y exactly.

We now choose the ansatz space to be Lagrange finite elements of order $p = 5$ for each component. As a solver for the discrete linear problem, we use a W(2,2)-multigrid cycle with an overlapping block Gauss-Seidel smoother where the blocks are associated with the vertices and consist of all degrees of freedom whose geometric objects (cell, edge, and vertex in 2D) share a given vertex. This smoother can be shown to be robust with respect to the discretization order, see [13], and leads to convergence rates below 10^{-2} in our test case. We obtain the results from Table 1 on a PC with a Pentium 4 (2.4 GHz) CPU and 1 GByte of RAM using CMUCL [3]. The table shows the values of a sample component $\hat{A}_{11}^{11}$ of the effective tensor which has 16 components of which only six are independent due to (2). We see that we achieve an accuracy of eleven digits in about two minutes. The eight components of the solution N_q^{lr} to (5) are shown in Figure 2. Obviously, it would have been possible to reduce the computational time significantly by exploiting symmetry, but we do not follow this path here.

Finally, we show the result for solving the analogous problem in $d = 3$ dimensions. Here, N_q^{lr} consists of 27 components and the effective tensor $\hat{A}$ has 81 entries of which only 21 are independent due to symmetry. Here, we choose Lagrange finite elements for $p = 4$ and use a block Gauss-Seidel smoother inside a multigrid algorithm where the blocks are defined by degrees

Cells	Unknowns	Matrix entries	Time (s)	$\hat{A}_{11}^{11}$
4	872	18836	3.2	$\underline{1.792}8477139$
16	4224	78780	10.6	$\underline{1.7925}713781$
64	17336	314716	34.3	$\underline{1.7925694}507$
256	69168	1256636	136.6	$\underline{1.7925694414}$
1024	275240	5022076	597.0	1.7925694414

Table 1: Results for computing a 2D elasticity tensor.

Cells	Unknowns	Matrix entries	Time (s)	$\hat{A}_{11}^{11}$
6	11583	660537	66.8	$\underline{2.62}61173098$
48	99522	5987691	465.4	$\underline{2.6231}943967$
384	783405	48023721	2960.7	2.6231430695

Table 2: Results for computing a 3D elasticity tensor.

of freedom belonging to geometric objects in the mesh. This smoother is not robust with respect to p, but nevertheless more efficient than the Vanka smoother on these rather coarse meshes. In this case, we can compute the effective tensor with five-digit accuracy in about eight minutes.

7 Discussion

How well does FEMLISP satisfy the requirements posed in Section 2? Arbitrary domains in arbitrary dimensions can be handled, and approximations of arbitrarily high order are also possible, though the step towards locally varying orders has still to be taken. Local mesh adaptivity and multigrid are available, as is algebraic multigrid. Since it is written in Common Lisp, FEMLISP is automatically interactive, and its source code comprises only about 20.000 lines at the moment, including documentation, testing and demonstration utilities.

However, two basic requirements have not yet been met satisfactorily. First, I have not addressed parallelization up to now. Second, FEMLISP is still very slow for several standard situations. This may seem astonishing, because it is indeed possible to make Lisp code run almost as fast or even faster than C code, see e.g. [12]. But, as pointed out in many places, for Lisp this involves an additional optimization effort which also tends to make the code inflexible if it is not done properly. Furthermore, it needs profiling on interesting benchmark problems which I hope to obtain now when FEMLISP is released to the public.

Which road will FEMLISP take from now on? There are two directions which do not pose any fundamental obstacles and on which I will work in the immediate future. One direction is implementing hp-adaptivity, the other is augmenting FEMLISP with strategies for solving nonlinear and time-dependent problems. While carrying out these improvements, I also hope to resolve the efficiency problem mentioned above.

How will FEMLISP look in the more distant future? This depends very much on what resonance FEMLISP receives from other researchers throughout the world. The fact is that, up to now, the numerical analysis community has more or less ignored Common Lisp, mainly for reasons which were valid 15 years ago, but are not valid any more today. I hope that FEMLISP will be a reason to reconsider this decision. I think that scientific computing is facing very difficult challenges right now, and it is simply careless to ignore a powerful tool like Common Lisp on the basis of antiquated beliefs. I admit that, at the moment, FEMLISP is still very young and can compete with established finite element software only for special applications. Nevertheless, I expect this to change rather soon.

Acknowledgements

This article was written while I was a research assistant in Heidelberg with the Technical Simulation group of Prof. G. Wittum, whom I want to thank for giving me the opportunity to work on this non-standard topic. I also want to thank him and the University of Heidelberg for allowing me to distribute FEMLISP under a free software license. Further, I am indebted to the interesting and entertaining *comp.lang.lisp* newsgroup as well as the CMUCL mailing list for much information from the year 2000 onwards.

References

[1] N. Bakhvalov and G. Panasenko. *Homogenization: Averaging Processes in Periodic Media.* Kluwer, Dordrecht, 1989.

[2] J. Bey. Simplicial grid refinement: On Freudenthal's algorithm and the optimal number of congruence classes. *Numer. Math.*, 85:1–29, 2000.

[3] CMUCL. Homepage. http://www.cons.org/cmucl.

[4] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical report, Computer Science Department, University of Tennessee, 1998.

[5] Data Explorer. Homepage. http://www.ibm.com/opendx.

[6] R. Fateman. Software fault prevention by language choice: why C is not my favorite language. *http://www.cs.berkeley.edu/~fateman/software.pdf*, (unpublished).

[7] R. Fateman, K. A. Broughan, D. K. Willcock, and D. Rettig. Fast floating-point processing with Common Lisp. *ACM Trans. on Math. Software*, 21:26–62, 1995.

[8] Femlisp. Homepage. http://www.femlisp.org.

[9] S. E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[10] G. Kiczales, J. Des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[11] Matlisp. Homepage. http://matlisp.sourceforge.net.

[12] N. Neuss. On using Common Lisp in scientific computing. In *Proceedings of the CISC 2002*. Springer-Verlag, 2002.

[13] N. Neuss. *Fast computation of effective coefficients with an interactive finite element tool box.* Habilitation thesis. Universität Heidelberg, Heidelberg, 2003.

[14] P. Norvig. *Principles of Artificial Intelligence Programming.* Morgan Kaufmann Publishers, Inc., San Francisco, USA, 1992.

[15] U. Rüde. *Mathematical and Computational Techniques for Multilevel Adaptive Methods*, volume 13 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, 1993.

[16] J. W. Ruge and K. Stüben. Algebraic multigrid (AMG). In S. F. McCormick, editor, *Multigrid Methods*, volume 3 of *Frontiers in Applied Mathematics*, pages 73–130. SIAM, Philadelphia, PA, 1987.

[17] K. Stüben. A review of algebraic multigrid. *J. Comput. Appl. Math.*, 128:281–309, 2001.

Preprintreihe IWR 2003

2003 - 01	(SFB 359) B. Cockburn, G. Kanschat, D. Schötzau The Local Discontinuous Galerkin Method For Linear Incompressible Fluid Flow: A Review
2003 - 02	(SFB 359) S. Bönisch, V. Heuveline, P. Wittwer Adaptive boundary conditions for exterior flow problems
2003 - 03	(SFB 359) R. Wehrse, B. Baschek, W. von Waldenfels The diffusion of radiation in moving media IV. Flux vector, effective opacity, and expansion opacity
2003 - 04	(SFB 359) I. Surovtsova Application of the one-dimensional model for blood flow through vascular prosthesis
2003 - 05	(SFB 359) E. Kostina, O. Kostyukova A Primal-Dual Active-Set Method for Convex Quadratic Programming
2003 - 06	(SFB 359) C. Surulescu On the Stationary Motion of an Incompressible Fluid Flow Through an Elastic Tube in 3D
2003 - 07	O. Sterz Multigrid for Time-Harmonic Eddy Currents without Gauge
2003 - 08	M. Dettweiler, S. Reiter On the middle convolution
2003 - 09	(SFB 359) V. Heuveline, R. Rannacher Duality-based adaptivity in the *hp*-finite element method
2003 - 10	(SFB 359) A. Renner Anisotropic Diffusion in Riemannian Colour Space
2003 - 11	(SFB 359) N. Neuß FEMLISP - a tool box for solving partial differential equations with finite elements and multigrid

Verfügbar unter der URL: `http://www.iwr.uni-heidelberg.de/sfb/Preprints.html`